游易-程序主题分享合辑

手游内存优化——以G6为例

李钊 2015.08.17 13:27 2097 10 4 查看原文

> 移动平台对于游戏的内存占用有较严格的要求，本文以G6为例，主要描述了基于cocos2d-x的2D游戏内存优化的一些工具与方法

十万个女妖精（G6）是一款弹幕射击类的2D手游，引擎为cocos2d-x 2.1，脚本采用Lua。弹幕射击类游戏属于重体验的游戏，对于游戏的流畅性有较高的要求，需要具有较高的帧率，这样才有可能获得良好的操作手感，同时项目定位于单机游戏，需要面对较多的中低端机型，支持的最低机型为iPhone 4或同档次机型，对于游戏的内存占用也有较高的要求。针对项目的这些特点，我们制定了一些针对性的内存优化策略。

1. 引擎的内存优化

（1）工具

针对Android的系统级profile工具并不多，但是对于IOS，我们可以选择Instruments帮助我们进行各种各样的优化。Instruments的Allocations和Leaks这两个profile项目，可以帮助我们监控引擎的内存分配，并发现其中某些可能的内存泄露。

图1

Allocations可以帮我们监控游戏在运行时刻的内存分配和总体占用情况，如图2，窗口上部显示了内存占用的波动情况，下部显示了内存的具体分配情况。

占用的统计方式也有好多种，例如Allocation Summary（图3）统计了各种不同种类的分配的总体情况。对于各种不同的类别，还可以分别查看详细的具体分析的情况，如图2的下半部分，在这里还会列出引用内存分配的具体函数名。

还有一种比较有用的统计方式是按函数的Call Tree来显示内存分配情况。如图4中，可以看到从start开始逐层调用所产生的内存占用。

Leaks是专门用来统计内存泄露的profile项目，如图5，上部显示了产生内存泄露的时刻，下部的窗口则显示了一些profile产生的结果，同时还会显示发生泄露的内存分配的调用者。图6的profile结果中显示了G6的getDeviceModelStr()和CCTextureFieldTTF的构造函数中可能存在存泄露。

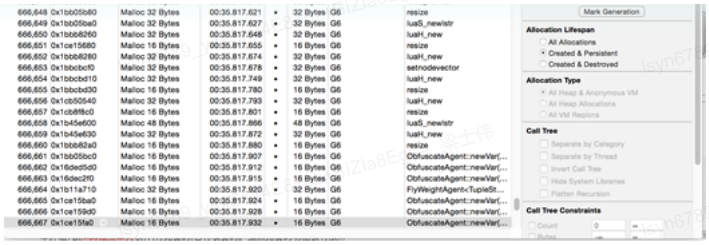


图2

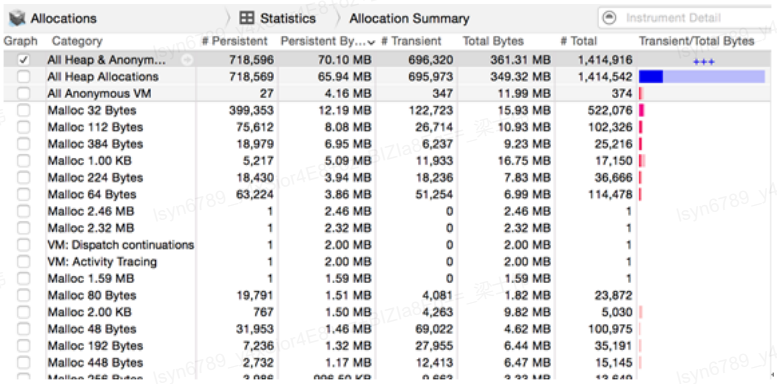

Allocations
Statistics
Allocation Summary
Instrument Detail
Graph Category # Persistent Persistent By... # Transient Total Bytes # Total Transient/Total Bytes
All Heap & Anonym... 718,596 70.10 MB 696,320 361.31 MB 1,414,916
All Heap Allocations 718,569 65.94 MB 695,973 349.32 MB 1,414,542
All Anonymous VM 27 4.16 MB 347 11.99 MB 374
Malloc 32 Bytes 399,353 12.19 MB 122,723 15.93 MB 522,076
Malloc 112 Bytes 75,612 8.08 MB 26,714 10.93 MB 102,326
Malloc 384 Bytes 18,979 6.95 MB 6,237 9.23 MB 25,216
Malloc 1.00 KB 5,217 5.09 MB 11,933 16.75 MB 17,150
Malloc 224 Bytes 18,430 3.94 MB 18,236 7.83 MB 36,666
Malloc 64 Bytes 63,224 3.86 MB 51,254 6.99 MB 114,478
Malloc 2.46 MB 1 2.46 MB 0 2.46 MB 1
Malloc 2.32 MB 1 2.32 MB 0 2.32 MB 1
VM: Dispatch continuations 1 2.00 MB 0 2.00 MB 1
VM: Activity Tracing 1 2.00 MB 0 2.00 MB 1
Malloc 1.59 MB 1 1.59 MB 0 1.59 MB 1
Malloc 80 Bytes 19,791 1.51 MB 4,081 1.82 MB 23,872
Malloc 2.00 KB 767 1.50 MB 4,263 9.82 MB 5,030
Malloc 48 Bytes 31,953 1.46 MB 69,022 4.62 MB 100,975
Malloc 192 Bytes 7,236 1.32 MB 27,955 6.44 MB 35,191
Malloc 448 Bytes 2,732 1.17 MB 12,413 6.47 MB 15,145

图3

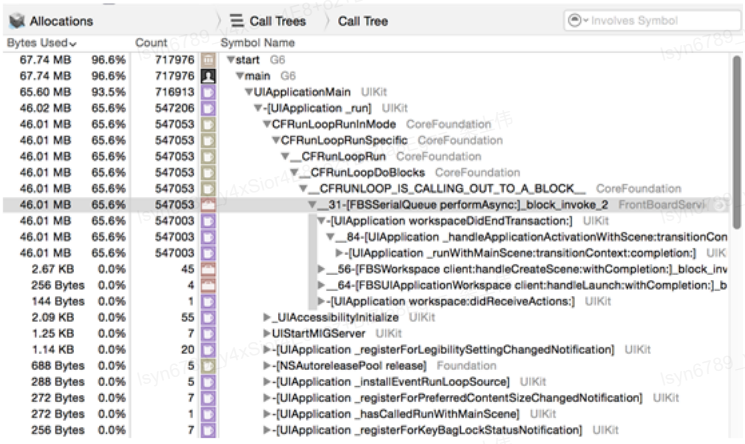

Allocations
Call Trees
Call Tree
Involves Symbol
Bytes Used Count Symbol Name
67.74 MB 96.6% 717976 start G6
67.74 MB 96.6% 717976 main G6
65.60 MB 93.5% 716913 UIApplicationMain UIKit
46.02 MB 65.6% 547206 -[UIApplication _run] UIKit
46.01 MB 65.6% 547053 CFRunLoopRunInMode CoreFoundation
46.01 MB 65.6% 547053 CFRunLoopRunSpecific CoreFoundation
46.01 MB 65.6% 547053 __CFRunLoopRun CoreFoundation
46.01 MB 65.6% 547053 __CFRunLoopDoBlocks CoreFoundation
46.01 MB 65.6% 547053 __CFRUNLOOP_IS_CALLING_OUT_TO_A_BLOCK__ CoreFoundation
46.01 MB 65.6% 547053 __31-[FBSSerialQueue performAsync:]_block_invoke_2 FrontBoardServi
46.01 MB 65.6% 547003 -[UIApplication workspaceDidEndTransaction:] UIKit
46.01 MB 65.6% 547003 __84-[UIApplication _handleApplicationActivationWithScene:transitionCon
46.01 MB 65.6% 547003 -[UIApplication _runWithMainScene:transitionContext:completion:] UIK
2.67 KB 0.0% 45 __56-[FBSWorkspace client:handleCreateScene:withCompletion:]_block_inv
256 Bytes 0.0% 4 __64-[FBSUIApplicationWorkspace client:handleLaunch:withCompletion:]_b
144 Bytes 0.0% 1 -[UIApplication workspace:didReceiveActions:] UIKit
2.09 KB 0.0% 55 _UIAccessibilityInitialize UIKit
1.25 KB 0.0% 7 UIStartMIGServer UIKit
1.14 KB 0.0% 20 -[UIApplication _registerForLegibilitySettingChangedNotification] UIKit
688 Bytes 0.0% 5 -[NSAutoreleasePool release] Foundation
288 Bytes 0.0% 5 -[UIApplication _installEventRunLoopSource] UIKit
272 Bytes 0.0% 7 -[UIApplication _registerForPreferredContentSizeChangedNotification] UIKit
272 Bytes 0.0% 1 -[UIApplication _hasCalledRunWithMainScene] UIKit
256 Bytes 0.0% 7 -[UIApplication _registerForKeyBagLockStatusNotification] UIKit

图4

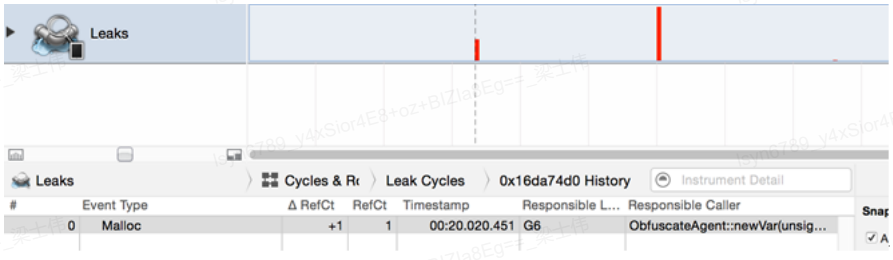
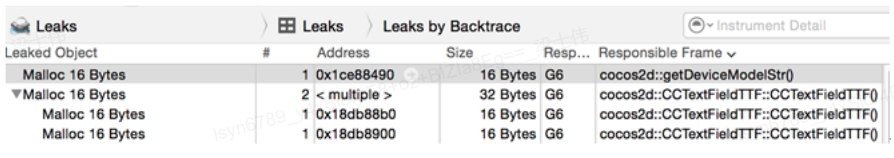

Leaks
Leaks
Cycles & R
Leak Cycles
0x16da74d0 History
Instrument Detail
Event Type Δ RefCt RefCt Timestamp Responsible L... Responsible Caller
0 Malloc +1 1 00:20.020.451 G6 ObfuscateAgent::newVar(unsig...

图5


Leaks
Leaks
Leaks by Backtrace
Instrument Detail
Leaked Object # Address Size Resp... Responsible Frame
Malloc 16 Bytes 1 0x1ce88490 16 Bytes G6 cocos2d::getDeviceModelStr()
Malloc 16 Bytes 2 < multiple > 32 Bytes G6 cocos2d::CCTextFieldTTF::CCTextFieldTTF()
Malloc 16 Bytes 1 0x18db88b0 16 Bytes G6 cocos2d::CCTextFieldTTF::CCTextFieldTTF()
Malloc 16 Bytes 1 0x18db8900 16 Bytes G6 cocos2d::CCTextFieldTTF::CCTextFieldTTF()

用计数，引用计数初始为1，每一次调用retain()时引用计数就加1，每次调用release()时，引用计数就减少1，当引用计数变为0时，就会触发对象的回收。使用autorelease()可以将对象加入CCAutoreleasePool库中进行管理。在每一个渲染帧的最后，引擎会尝试release一次CCAutoreleasePool中的每一个对象，如果没有其他的对象在引用它，那么这个对象就会被销毁。

```
class CC_DLL CCObject : public CCCopying
{
protected:
    // count of references
    unsigned int        m_uReference;
    // count of autorelease
    unsigned int        m_uAutoReleaseCount;
```

使用这套机制管理的引擎对象不太会出现内存泄露的情况，但是对于一些有需要在脚本进行手动retain和release操作的对象则非常容易出现由于引用计数的问题而产生的内存泄露问题。基于这套机制，我们向引擎添加了一些统计数据的代码，统计一些常用类的对象计数，在对象在构造时，对应类的统计计数就会加1，在对象析构时，统计计数就会减1。例如对于CCSpriteFrame对象，我们进行如下的修改：

```
int CCSpriteFrame::ccspriteframe_node = 0;

#ifdef _WIN32
CCSpriteFrame::CCSpriteFrame()
{
    ccspriteframe_node++;
}
#endif

int CCSpriteFrame::getCount()
{
    return ccspriteframe_node;
}

CCSpriteFrame::~CCSpriteFrame(void)
{
#ifdef _WIN32
    ccspriteframe_node--;
#endif
    CCLOGINFO("cocos2d: deallocing %p", this);
    CC_SAFE_RELEASE(m_pobTexture);
}
```

在游戏运行的一些关键节点(例如进出副本时)，向引擎查询这时引擎内各种对象的计数，就可以很容易得出哪些对象产生了泄露。同时，在这些关键节点，通过查询引擎内几个重要的缓存池，例如CCTextureCache、CCSpriteFrameCache中对象的引用计数，获得对应资源的名称，更可以反推脚本中的哪些逻辑存在问题，造成了内存泄露。

上面的方法虽然简单，却可以有效地找出cocos2d-x引擎中的内存泄露点。

（3）纹理内存占用优化

纹理是2D游戏内存占用的一个主要方面。很多时候，造成游戏crash的原因就是美术资源大量使用了帧动画，造成纹理的内存占用居高不下。如何减少这一部分的内存占用，在内存紧张，CPU空余的时候，除了说服美术尽量使用2D骨骼动画等其他方式来制作游戏资源的同时，选择合适的纹理格式也是很重要的问题。

纹理格式总的说来，可以分为GPU纹理和非GPU纹理两种，GPU是指可以直接被显卡接受的纹理格式，例如windows上的dds，ios上的pvr，android上的etc这些格式。非GPU纹理是指需要进行读取、解压等操作后才能提交给GPU进行渲染的格式，例如常见的PNG、JPG等格式，GPU纹理往往拥有显存占用小、纹理加载快的优点。下面的表格中是cocos2d-x支持的一些常见的纹理格式。此外还存在ATITC(ATI texture compression) 、S3TC(S3 texture compression) 等压缩，但是它们目前为止还并不为大多数机型所支持，所以不在我们的考虑范围内。从表格中可以发现，如果我们在android上使用ETC，在IOS上使用PVR就有可能达到节约纹理内存占用的目的。

格式	是否GPU直接支持格式	每像素内存占用	支持的平台
PNG	否	32bit	Android,IOS
PVR	是	2bit~32bit	IOS，以及部分使用PowerVR显卡的安卓机
ETC	是	4bit	Android

(i) 关于PVR格式

PVR(PowerVR texture compression)，是由Imagination PowerVR 开发的纹理格式，用于PowerVR生产的移动GPU上。到目前为止，所有IOS设备的显卡均为PowerVR生产，所以在IOS上使用PVR格式的纹理是一个不错的选择，PVR按是否为压缩纹理主要又有如下的几种：

格式	每像素内存占用	压缩纹理	Alpha通道	显示效果	图片尺寸
PVR	32bit	否	支持	好	支持NPOT
PVRTC2	2bit	是	支持	差	边长最小8像素，且必须是方的

(1) PVRTexTool（旧版，官方已下载不到）

(2) xcode提供的texturetool

(3) PVRTexTool（新版）

(4) TexturePacker

（ii）关于ETC格式

ETC（Ericcson texture compression ）目前又分别ETC1与ETC2，如下表：

格式	每像素内存占用	压缩纹理	Alpha通道	显示效果	标准支持
ETC1(PKM)	4bit	是	不支持	尚可	OpenGL ES 2.0扩展
ETC1(KTX)	4bit	是	不支持	尚可	OpenGL ES 2.0扩展
ETC2	4bit~8bit	是	支持	尚可	OpenGL ES 3.0

ETC1虽然不是OpenGL ES 2.0必须要支持的格式，但是在实践中，在我们的目标Android机型中没有发现不支持ETC1的，Alpha通道则可以通过其他方式来进行支持。由于ETC1显示的半透明的纹理效果较PVRTC4更好，同时iPhone的GPU虽然在技术文档的层面上应该也支持ETC1，但是经过我们的实际测试，发现在IOS上，ETC1格式的纹理只能通过软解来支持，对渲染速度也有较大的影响，因此，我们放弃了在IOS上使用ETC1格式的想法。

我们还对ETC1格式的POT与NPOT的支持情况进行了测试，发现较新的机型支持NPOT没有问题，但是像三星的S2等机型，对于NPOT格式的ETC1纹理会发生闪退的情况，于是决定使用POT的ETC1纹理。

由于原生的ETC1不支持Alpha通道，因此需要扩展，扩展方式主要有两种，一种是将alpha通道扩展到原图的下方，形成一个和原图一样大小的区域来表示原图中的alpha信息，对于PKM和KTX可以使用这种方式，另一种方法是将alpha信息写入KTX的meta信息中，在读入内存时，从meta中取出alpha信息，生成一张与原图一样大小的A8纹理，两种方式都会增加一倍的内存占用，所以经过扩展的ETC1每像素大约会占8bit的内存。

由于ETC1和PVRTC4对于纹理的尺寸都有比较严格的要求，贴图进行格式转换时，需要将图片扩展成方的且边长是2的幂（对于PVRTC4），或者边长是2的幂（对于ETC1）,这样有可能造成较大的内存浪费，于是有必要进行贴图的合并，将一系列小图转换成大图（如图7）。对于简单使用画家算法进行渲染的cocos2d-x来说，这样还可以有效地节约GPU的像素填充率。

图7

综合考虑一些其他因素，我们将压缩纹理优化的技术规范定为：

- IOS：PVRTC4，纹理是POT，且长宽相等，最大为2048*2048
- Android：PKM，纹理是POT，最大为2048*2048（其中一半是原始图，一半是alpha信息）
- 一些压缩过后显示效果较差的图片，按情况选择PNG8或者原始PNG

为了能够方便策划灵活地对图片的压缩格式进行修改，我们开发了一个工具，策划只需要填表就可以指定某张图的压缩格式。

2. 脚本的内存优化

（1）脚本内存的profile工具

Lua脚本的优化主要使用了snapshot工具，snapshot可以打印出某个时刻，内存中的脚本对象，在游戏内的两个关键的时刻（例如进出游戏时），分别调用一次snapshot（每次之前都要进行一次full gc）得到脚本内存的一个镜像，将这两次记录下来的内存求一个差值，去掉共有部分，就可以得到这段时间内脚本新产生并且没有回收的对象，如在G6中一次内存profile的结果，两次得到的差值如图8：

图8

从图8的左边我们可以知道，0645F6A8是entity.lua中的entltyList变量，entityList中key为1234的项对应的value是userdata 0845F630，0845F630又包含一个成员叫_aoi_id_list，由于代码中_aoi_id_list是clsCombat类的成员，所以0845F630是一个clsCombat的实例，于是就可以整理得到右边的一个引用关系图。entityList是在entity.lua内部一个用来记录游戏中所有entity的一个列表，entity销毁以后，应该在release方法中销毁引用，不应该再存在这样的引用关系，于是我们就可以把这个内存泄露的原因定位到clsCombat没有被正常销毁，或者是release的代码存在问题。

在snapshot的帮助下，脚本内可以稳定重现的内存泄露是比较容易发现和定位的。

（2）基于时间片的GC

对于Lua这种自动管理内存的脚本语言，GC是一个不得不面对的问题。目前主流编程语言的GC主要有引用计数，或者标记-清除、分代收集这两种。因为引用计数不能解决循环引用的问题，Python就两种兼而有之，而Lua则只有标记-清除、分代收集这一种GC方法。

传统的标记-清除（Mark-and-Sweep）算法是一个两阶段的算法：

（i）Mark phase(标记阶段)

1）每个可被gc的对象都拥有一个标志位，初始为unmarked

2）定义程序中第一层可访问的对象集合为根对象集合

3）递归遍历根对象集合中所有对象的引用关系，将标记为unmarked的对象标记为marked

（ii）Sweep phase(清除阶段)

遍历所有现存的对象：将标记位还是unmarked的对象释放，同时将标志为marked的对象重新标志为unmarked，为下次gc做准备。

Lua对这一算法进行了改进，引入：old white: 老对象初始状态；current white：新对象的初始状态；gray：待变黑色状态；black：老变量继续存活的状态，这几种颜色之一来表示某个对象的当前状态，并以old white和current white来处理gc过程中产生的新对象。Lua的GC是一个增量式的过程，由一个一个step来完成，在若干个step中完成标记-清除算法的各个阶段。

由于内存中对象数量庞大，完整地进行一次GC算法需要很长的时间，因此会带来游戏内的明显卡顿。

弹幕游戏一个明显的特征就是子弹很多，游戏内每秒种都在产生大量的对象，不开GC，很快就会由于内存爆满而造成游戏闪退。对于G6来说，由于存在一些历史原因，子弹数据结构，某些还带有AI，用cache的方式重用对象时，经常很难使对象能很好的回到初始状态，子弹池也不是一个很好的解决方案。然而弹幕游戏对于流畅度要求高，因此必须解决GC带来的卡顿问题。

Lua每一次GC的完成，都不能给出一个明确的时间界，不能在GC花费的时间达到某个值时主动停止GC。不过我们认为一个singleStep的时间花费是不多的，因此我们在此基础上实现了一个基于时间片的GC，使得GC花费的时间基本可控，核心代码如图9，可以指定每个时间片GC最多执行的时间，甚至可以根据脚本逻辑的繁忙程序来动态调整GC的花费。同时需要完全关闭Lua的GC，将GC的触发时机完全交由脚本的GC模块来处理，脚本GC模块的逻辑如下：

```
double luaC_step_slice(lua_State *L, double _time_limit ) {
  double time_limit = _time_limit;
  double time_during = 0;
  global_State *g = G(L);
  g->gcdept += g->totalbytes - g->GCthreshold;
  do {
    double s, e, delta;
    int i;
    s = lua_gettime();

    for(i=0;i<5;i++) {
        singlestep(L);
        if (g->gcstate == GCSpause)
          break;
    }

    e = lua_gettime();
    delta = e - s;
    time_during += delta;
    if (g->gcstate == GCSpause)
      break;
  } while (time_during < time_limit);
  return time_during;
}
```

图9

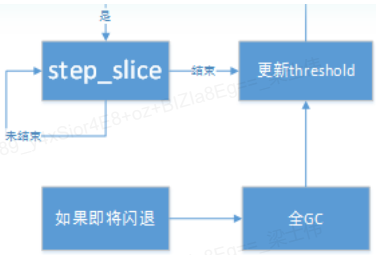


图10

如此处理之后，由于无法保证GC的速度比内存分配的速度快，因此还需要在IOS的applicationDidReceiveMemoryWarning和Android的OnLowMemory回调中触发一次full gc，并相应回收一些缓存。

G6在使用基于时间片的策略之后，感觉不到由于GC带来的卡顿，在instruments上实测GC在iPhone 4上的开销大约占总时间的2%

3. 总结

使用上面的一些优化方法，同时结合的模块的Lazy Import改进，美术资源的清理，脚本缓存的定时释放等优化方式，我们最终使G6的内存占用得到了有效的控制。

全部评论 4

请输入评论内容

还可以输入 500 个字

(可添加1个视频+5张图片) 匿名 评 论

最热 最新

匿名

4楼 赞

2016-08-12 17:40 回复 0

rocky(王建军)

3楼 赞

2016-02-18 13:16

匿名

1楼 赞

2015-08-28 10:35 回复 0

加载完毕,没有更多了